The Dining Philosophers with Pthreads

Dr. Douglas Niehaus
Michael Jantz
Dr. Prasad Kulkarni

Introduction

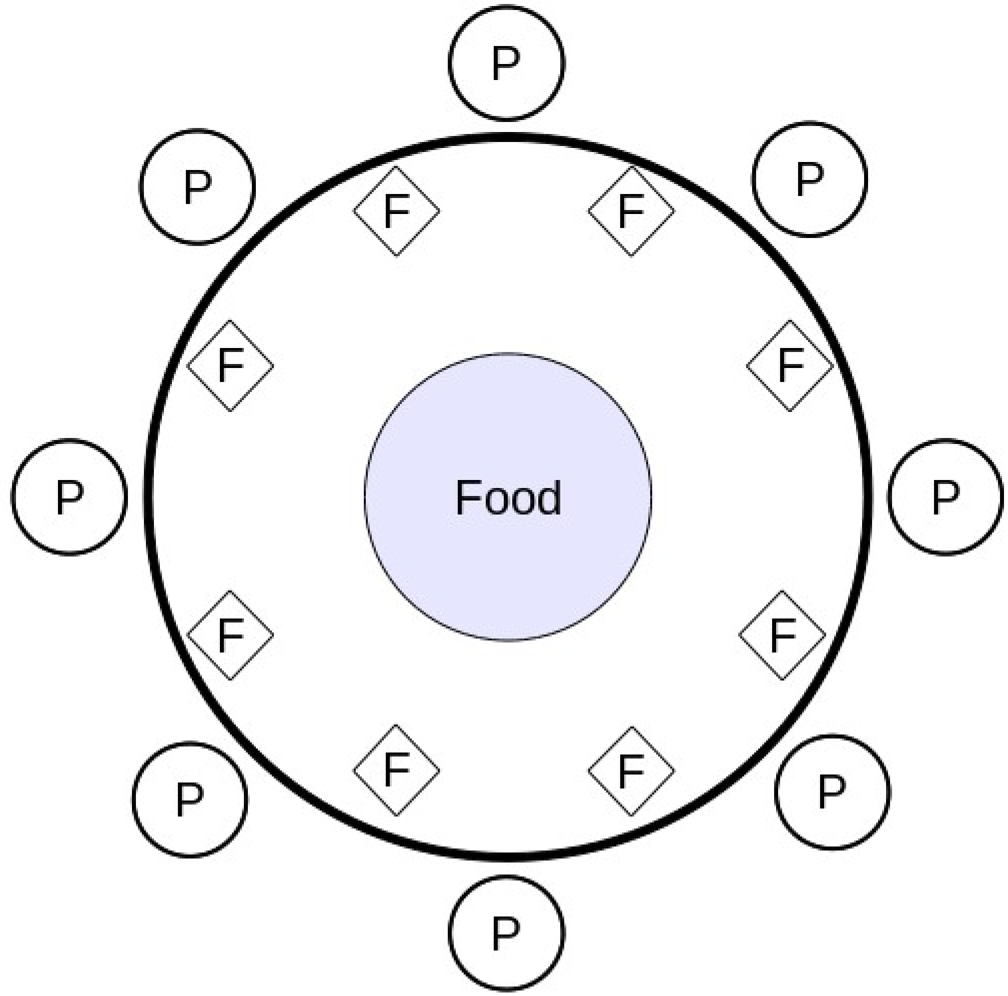
- The Dining Philosophers canonical problem illustrates a number of interesting points about concurrency control that recur in various situations
  - Multiple threads using multiple resources
  - Different sets of resources used by different threads
  - Threads spend different amounts of time using resources and between intervals of resource use
  - Deadlock can occur because of a set of interactions among different threads and resources
- First proposed by Djikstra (1965) as a problem of coordinating access by five computers to five tape drives
  - Retold in its more amusing current form by Hoare
- Few real-world problems map directly onto its structure
  - But many share characteristics: multiple threads, multiple resources, varied patterns of resource use

Dining Philosophers

- A set of philosophers spend their lives alternating between thinking and eating
- Philosophers sit around a table with a shared bowl of food
- To eat, philosophers must hold two implements
- Implements are placed on the table between philosophers
  - Each philosopher this has a right and left implement
  - Each philosopher uses a different set of resources
- Implements can only be acquired one at a time
- When a philosopher becomes hungry, she tries to pick up the left implement and then the right
- If an implement is missing, the philosopher waits for it to appear
- A hungry philosopher holding two implements eats until no longer hungry, puts down her implements and thinks

Dining Philosphers

- N philosophers, N fork
- Food has unrestricted concurrent access
- Forks are exclusive use resources
- Each fork plays a differ role for its philosophers (L/R)
- Each fork used by a different set of philosophers
- Deadlock appears quite unlikely to happen
- Happens “quickly” in practice



Pthreads Implementation

- Starter code implements the “classic” dining philosophers problem with its vulnerability to deadlock
- Assumes familiarity with Pthreads concepts in previous labs
  - Concurrent execution of Pthreads
  - Mutex used for mutual exclusion
  - Condition variable use for signal-wait interaction
- Starter code also contains some components labeled ASYMMETRIC and WAITER which are associated with two different approaches to a solution you will work on.
- Go ahead and unpack the starter code and run the current implementation

```
bash> tar zxvf eecs678-pthreads_dp-lab.tar.gz
```

Pthreads Implementation

- Code is a fairly straightforward implementation decomposed into a number of components
  - dining_philosophers.c
- Code begins with includes and defined constants
  - Constants are used to control many aspects of behavior
- Next, a definition of the *philosopher* structure
  - Note the *prog* and *prog_total* fields which track the number of times a philosopher has gone through the think-eat cycle during an accounting period and during program execution, respectively
- Next com some global variables:
  - *Diners*: array of philosopher structures
  - *Stop*: global stop flag
  - *chopstick*: array of mutexes representing the chopsticks

Pthreads Implementation

- Global continued
  - *waiter*: mutex used to represent the waiter the waiter-based solution
  - *available_chopsticks*: array of integers used to represent chopstick availability in the waiter solution
- Next is a set of utility routines used in various solutions
  - Return pointers to philosopher to left and right of argument, chopstick to left and right, and pointer to available flag of left and right chopstick of a given philosopher
- *think_one_thought*( ) and *eat_one_mouthful*() routines
  - Used in *dp_thread*( ) routine to represent activity
- *dp_thread( )* routine is code executed by each philosopher thread which implements the think-eat cycle until told to stop, and does accounting on how many cycles completed

Pthreads Implementation

- *set_table( )* routine initializes data structures representing chopsticks, initializes the philosopher structures and creates the philosopher threads
- *print_progress( )* prints progress statistics for each philosopher, and zeroes the prog filed so progress during each accounting period is counted as well as the total
  - Five philosophers per line and a blank line between statistics for each accounting period
- *main( )* calls *set_table( )*, prints out a header, and falls into the accounting and deadlock detection loop
  - Root thread zeroes philosopher period progress, then sleeps for ACCOUNTING_PERIOD seconds
  - Checks to see if any progress made while it slept
  - Infers deadlock if not, and sets Stop
  - Prints statistics in any case

Pthreads Implementation

- Run the existing code

  bash> cd pthreads_dp; make dp_test
- Your output should be similar, but remember thread behavior and deadlock are affected by many random factors
  - Context switches, other load on system, interrupts, etc

```
plato:starter_code$ make dp_test
gcc -g dining_philosophers.c -lpthread -lm -o dp
./dp

Dining Philosophers Update every 5 seconds
------------------------------------------
p0=   1012/1056   p1=      1/1   p2=    492/492   p3=    913/913   p4=      0/0

p0=      0/1056   p1=      0/1   p2=      0/492   p3=      0/913   p4=      0/0

Deadlock Detected
```

Asymmetric Solution

- Example output shows that deadlock occurred during the first accounting period, after threads had performed a variable number of think-eat cycles
  - “P1 = 123/456” entry indicates that P1 executed 123 think-eat cycles in the current accounting period and has 456 total
  - Numbers may not be completely consistent as there is no concurrency control between main and philosopher threads
  - Try running the test several times and see that behavior varies
- Deadlock occurs because each philosopher has picked up the left fork before any have pick up the right
  - Happens much more quickly than most people would expect
- Asymmetric solution is to have the even numbered philosophers pick up in left-right order, while odd-numbered pick up in right-left order

Asymmetric Solution

- Make a copy of dining_philosophers.c into dp_asymmetric.c and update the Makefile appropriately
- Make the necessary change to dp_thread where the string ASYMMETRIC appears in the comment: test *me*->*id* for even or odd and alter mutex lock order accordingly

  bash> make dp_asymmetric_test

- If your implementation is correct, then the program should run for 10 5-second cycles and complete without deadlock
- Note how many think-eat cycles each philosopher makes in each accounting cycle and total
  - This will vary with the platform (cycle4, 1005D-*, etc)
  - Was several hundred thousand on development machine
- Note that progress by each philosopher is roughly equal
- Try running it a few more times and see how much behavior varies due to random chance and system context

Asymmetric Solution

- All philosophers still randomly compete for their left and right chopsticks, holding their first and waiting for the second
- As long as thinking and eating periods vary randomly and other factors make when a philosopher tries to pick up their chopsticks vary randomly, then progress should be roughly equal and no philosopher should starve
- However, if a set of philosophers ever began to share the same “rhythm” then one philosopher might be at a disadvantage

Waiter Solution

- Now consider a slightly more complex solution using a Pthread condition variable approach
  - Mutex *waiter* represents a waiter in the cafe that will “give” the chopsticks to a philosopher as a *pair*
  - Note that this will constrain concurrency more than the asymmetric solution as this creates a region where only one philosopher at a time can obtain its chopsticks
- Copy dining_philosophers.c into dp_waiter.c
  - Look for “WAITER SOLUTION” in the code
  - Relevant changes are in dp_thread() code where philosophers obtain and give back their chopsticks
- This solution does not need the *chopstick* array of mutexes
  - Use the array of integers *available_chopsticks* instead, whose integrity will be protected by the *waiter* mutex, and condition variable programming pattern

Waiter Solution

- Get-chopsticks section ensures that testing my_chopsticks_free and mark_my_chopsticks_free set of operations are ATOMIC using *waiter*
- Free-chopsticks section uses waiter to ensures the mark_my_chopsticks_free and Signal sets of operations are done ATOMICALLY
- Consider types and pointers carefully as the helper routines return pointers to available flags and philosophers

```
pthread_mutex_lock(&waiter);

while (!( my_chopsticks_free )) {
    pthread_cond_wait(&(me->can_eat), &waiter);
}

mark_my_chopsticks_taken;
pthread_mutex_unlock(&waiter);

Eat;

pthread_mutex_lock(&waiter);

mark_my_chopstick_free;
Signal those who might care they became free

pthread_mutex_unlock(&waiter);
```

Waiter Solution

- When your solution is complete and correct, your solution should produce output similar to the asymmetric solution
  - Runs through 10 cycles and completes without deadlock
- Note, however, that the number of think-eat cycles is significantly lower
  - Why?
- Another point of interest is the while loop testing the condition and calling pthread_cond_wait()
  - Why does this need to be a loop
    - Hint: Consider possible events between when the decision to send the signal is made and when the signal is received

Waiter Solution

- Does this solution prevent starvation?
  - Hint: NO !!!
- Try to extend your solution to count the number of times a philosopher is awakened and both chopsticks are *not free*, so it must wait again
- Experiment with tests in the chopstick freeing area that send a signal to a philosopher only when both its chopsticks are free
  - You should find that a small but significant percentage of the time a chopstick is taken between when the signal is sent and when the receiving philosopher tries to get its chopsticks
- Consider what would happen in these retry cases if the *while* loop was an *if-then* instead

Conclusions

- The dining philosophers is a simple problem with a surprising number of subtle aspects
- Deadlock seems extremely unlikely, yet happens quit quickly
- Solutions are not all that difficult, but have different implications
- Plausible but incorrect solutions also easy to construct
- Shows that knowing if a solution is correct is also *hard*
- Neither of these solutions to preventing deadlock prevent starvation
- Consider how to implement the Waiter solution with a Monitor representing the waiter
  - Waiter can maintain a queue of requests, ensuring all philosophers eventually eat